Extracted from:

High Performance PostgreSQL for Rails

Reliable, Scalable, Maintainable Database Applications

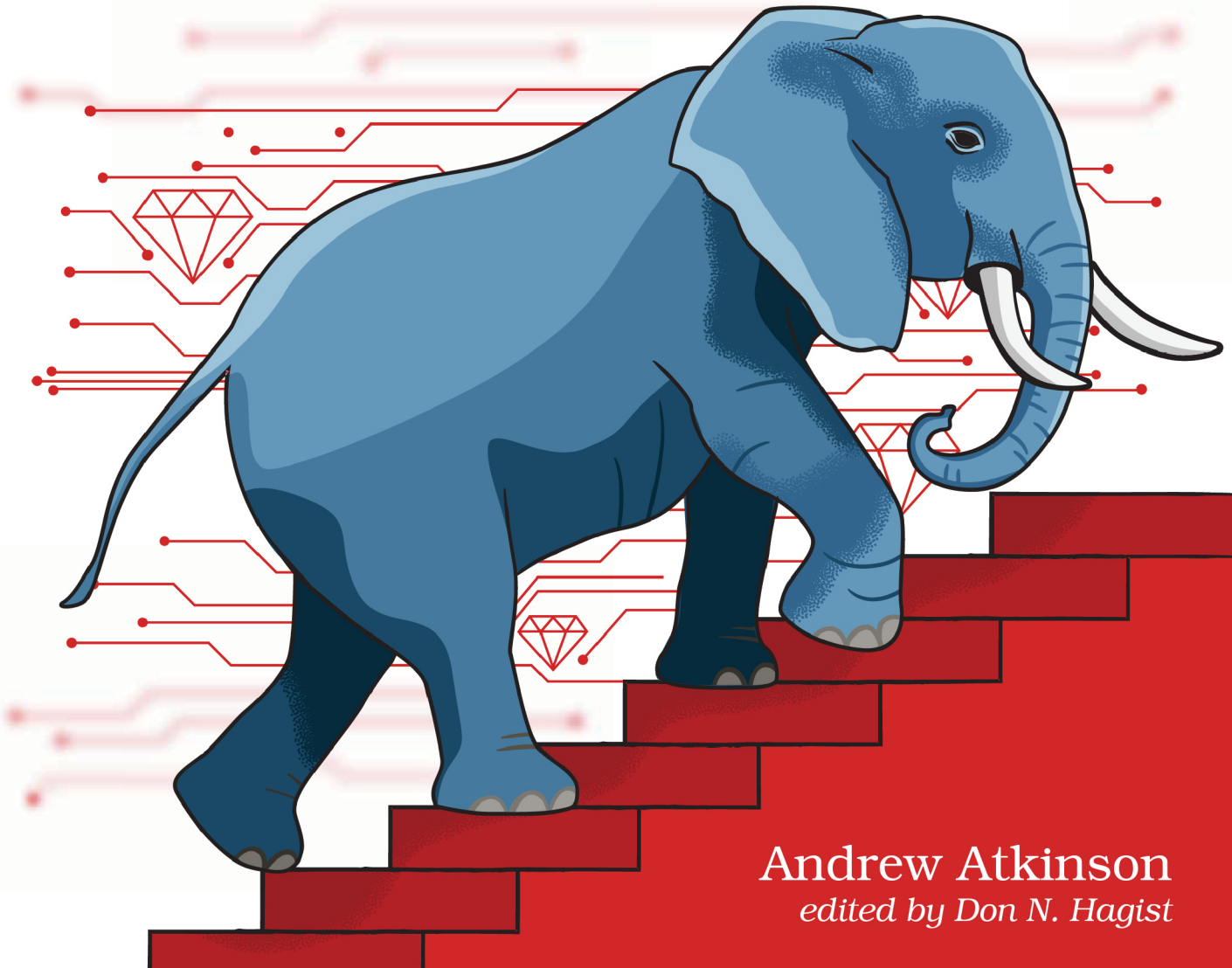
The Pragmatic Bookshelf

Dallas, Texas

The Pragmatic Programmers

High Performance PostgreSQL for Rails

Reliable, Scalable, Maintainable Database Applications



Andrew Atkinson

edited by Don N. Hagist

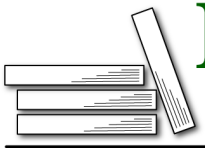
High Performance PostgreSQL for Rails

Reliable, Scalable, Maintainable Database Applications

Andrew Atkinson

The Pragmatic Bookshelf

Dallas, Texas



For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 979-8-88865-038-7
Encoded using the finest acid-free high-entropy binary digits.
Book version: B1.0—August 30, 2023

CHAPTER 7

Improving Query Performance

In this chapter you'll analyze SQL queries running on PostgreSQL, developing a richer understanding of their characteristics as they relate to performance.

To do that you'll learn how to generate and read query execution plans, and use statistics that the database continually collects. Once you understand which portions of a query are most costly, you can begin to use tactics that reduce the costly portions which can improve the overall scalability of your server instance.

Query optimization is a complex subject, with entire books dedicated to covering it. In this chapter you'll get equipped with the basics, learning where to find information and how to interpret it.

Review the following terminology to learn about it for the first time or get refreshed on it. You'll be working with terms like *selectivity*, *cardinality*, a variety of Index types, filtering, sorting, and more in this chapter.

Query Performance Terminology

- Selectivity — How selective a query is
- Cardinality — How many unique values there are
- Sequential Scan — Reading all rows in table
- Index Scan — Fetching values from an Index on a table

To get insights from your query execution, you'll also work with your logs. Before going to PostgreSQL logs, what kind of query information can you get from Active Record logs?

Logging Slow Queries With Active Support Notifications

With Ruby on Rails and without any extra gems, plugins, or database extensions, a wealth of query information can be added to the Rails application log. *Active Support Notifications*[1] can be used to monitor slow queries. (See: *Track slow DB queries without additional gems*[2])

To see this in action, read and understand the *Slow Query Subscriber* class added to Rideshare that uses Active Support.

The *Subscriber* listens for events formatted as sql.active_record that have query information. The duration of the query can be calculated using the start and finish times associated with the event. When the duration exceeds 1 second, the full SQL statement can be logged using the Rails logger.

Review the class in Rideshare or copy this:

ruby/slow_query_subscriber.rb

```
# Inspiration: https://twitter.com/kukicola/status/1578842934849724416
class SlowQuerySubscriber < ActiveSupport::Subscriber
  SECONDS_THRESHOLD = 1.0

  ActiveSupport::Notifications.subscribe('sql.active_record')
  do |name, start, finish, _, data|

    duration = finish - start

    if duration > SECONDS_THRESHOLD
      sql = data[:sql]
      Rails.logger.debug "[#{name}] #{duration} #{sql}"
    end
  end
end
```

Open bin/rails console to test this out. By running the query SELECT pg_sleep(1); from Active Record as a query that simulates taking 1 second or more, you'll see the instrumentation get triggered and the following log output.

```
irb(main):004:0> ActiveRecord::Base.connection.execute("SELECT pg_sleep(1)")
[sql.active_record] 1.005035 SELECT pg_sleep(1)
   (1005.2ms)  SELECT pg_sleep(1)
```

In this technique you've logged a slow query from the client application side. What about when you want to view slow queries from the server side?

1. https://guides.rubyonrails.org/active_support_instrumentation.html
2. https://twitter.com/kukicola/status/1578842934849724416/photo/1

Capture Query Statistics In Your Database

The queries in your database consume resources and it's helpful to focus your optimization efforts on the most costly queries. To do that you'll want to get a global view of all queries to see which ones are most costly. You'll use the pg_stat_statements[3] module which you configured in an earlier chapter, and will be referred to as PGSS.

PGSS takes each query and removes the parameters from it, creating a sort of query group or pattern. Each query then can be put into this group or pattern, and statistics can be collected at the group or pattern level. PGSS then makes the statistics available with a system catalog view.

These groups or patterns are called the *normalized* form of the query. Each normalized query gets a Query Identifier (queryid). The queryid uniquely identifies the original query text before the normalization process. The parameters are removed in the normalization process and replaced with placeholder characters.

Part of the purpose of the normalization where parameters are removed, is so that queries with the same structure can be grouped together. Statistics can then be computed across queries that are similar, sometimes referred to as a "group".

Query statistics are at the group level including the duration and the number of calls, across all instances of a query within the group that may have very different sets of specific parameters.

Enabling the PGSS module requires a database restart. PGSS is generally recommended to add for all databases. Although it adds a minimal amount of latency to collect statistics, the information it provides is worth it. PGSS is supported by cloud providers of PostgreSQL. If you don't already have it enabled in your production database, plan a time to make this parameter change and restart your database during a low activity period.

To configure the module, add pg_stat_statements to shared_preload_libraries in postgresql.conf. Once that's done and PostgreSQL has restarted, you'll enable the extension.

To enable the extension, run the following statement from psql. This only needs to be done once per database.

3. https://www.postgresql.org/docs/current/pgstatstatements.html

sql/create_extension.sql
```
CREATE EXTENSION pg_stat_statements;
```

You may also use ALTER SYSTEM to modify shared_preload_libraries from a psql prompt but this isn't recommended. This method generates a value in postgresql.auto.conf that overrides the value in postgresql.conf (Thanks to Lukas Fittl for this tip![4]).

This can lead to confusion about which value is active, so consider making all changes only in your PostgreSQL config file and keeping a version controller backup of your single config file.

Rideshare enabled PGSS by enabling the extension via a Rails Migration.

Now that you've enabled PGSS, read on to find out how to populate and review query statistics for Rideshare.

Rideshare Query Statistics

Since Rideshare isn't a running system, you'll need to simulate application activity by populating some query information.

To do that, start up a Rideshare server by running bin/rails server from your Terminal.

In another terminal window, run bin/rails simulate:app_activity to simulate activity. You'll should see queries being logged to your server. Since PGSS was enabled earlier, when those queries arrived in PostgreSQL, the denormalization and statistics collection process was happening in the background.

PGSS tracks 5000 normalized queries by default. To track more than that, raise the pg_stat_statements.max value. The least-executed queries are discarded when the max is reached.

From psql, query SELECT * FROM pg_stat_statements; to view everything PGSS has collected. A more useful query, though, might be to look at the top few slowest queries.

Once you've done that, PGSS should have captured some statistics.

From your terminal, run psql --dbname rideshare_development again and then use the following query to look at some of the queries and fields of data from PGSS. You may want to run SELECT pg_stat_statements_reset(); to reset the statistics and run the Rideshare simulation again to isolate the results to being only from the simulation.

4. https://www.postgresql.org/docs/current/config-setting.html

sql/ten_worst_queries.sql

```
SELECT
  total_exec_time,
  mean_exec_time,
  calls,
  query
FROM pg_stat_statements
ORDER BY mean_exec_time DESC
LIMIT 10;
```

You should now see some Rideshare queries in the PGSS results. A result row will contain total_exec_time, avg_ms, calls, and the normalized query like in this example:

sql/pg_stat_statements_result.sql

```
-- -[ RECORD 1 ]-------------------------
-- total_exec_time | 2903.9372489999996
-- avg_ms          | 580.7874497999999
-- calls           | 5
-- query           | SELECT "users".* FROM "users" WHERE "users"."type" = $1
```

This is great. You can now review statistics collected database-wide for Rideshare queries. You're accessing these results from psql.

When you work on a team of application developers who might not want to use psql to view this information, how can you more easily expose it to them?